Final Report
NASA Contract NAS5-31354

James A. Storer

Computer Science Department
Brandeis University
Waltham, MA 02254

phone: 617-736-2714
fax: 617-736-2741
email: storer@cs.brandeis.edu

August, 1994

Staff Involved in the Project

Faculty:

J. Storer
M. Cohn

Post-Doc: *J. Lin*

Ph.D. Students:

S. De Agostino
B. Carpentieri
C. Constantinescu

Current Research

We are continuing the work begun in Years 1 (1991 - 1992) and 2 (1992 - 1993) and reported in our earlier progress reports this year. The thrust of our group continues to be the study of on-line fully adaptive algorithms for data compression with real-time parallel implementations. Such algorithms are key to NASA applications where high speed is required and diverse data sets need to be handled.

Here we summarize what's new from what was reported last year.

- *Image Compression:* A paper on our basic single-pass adaptive VQ with variable size and shaped codebook entries has appeared in the *Proceedings of the IEEE.* A new paper was presented at the *1994 IEEE Data Compression Conference* that describes the use of KD-trees for a fast serial implementation that can run on a UNIX workstation. In addition, this paper describes a number of key improvements to the basic algorithm. The Computer Science Department at Brandeis University has recently received a 1 million dollar grant from the NSF for the purchase of parallel computing equipment; part of these funds have already been used to purchase a 4,096 processor MASS-PAR machine; the remainder was used to purchase a 16-node SGI Challenge machine. We have been conducting experiments with this machine on practical sub-linear parallel implementations of the algorithm.

- *Video Compression:* Our work on the basic adaptive displacement estimation algorithm that tracks variable shaped groups of pixels from frame to frame has appeared in the same issue of the *Proceedings of the IEEE* as our work on adaptive image compression. In addition, we have submitted for journal publication new work on the integration of this algorithm into a complete video and image sequence compression system. We are in the process of compiling extensive experimental results with the system.

- *Parallel Algorithms:* Our work on sublinear algorithms for parallel text compression has been submitted for journal publication. We have conducted experiments with our new approach to sub-linear text compression that closely approximates optimal compression but is much more practical to implement. Using an extremely simple parallel model (a linear array where processors can only talk to adjacent neighbors), we have achieved poly-log time and extremely close approximation to optimal compression. As parallel computers become more common, algorithms such as this will provide practical ways to fully utilize the power of these machine in NASA applications involving large amounts of data.

- *Error Propagation:* A paper on our basic error resilient algorithm has been submitted for journal publication. We are continuing our investigation of "error resilient" systems, and their application to lossy systems.

Appendix: As indicated above, the two papers that recently appeared in the *Proceedings of the IEEE* give good summaries of the key work performed under this contract. Attached are copies of these papers.

Improved Techniques for Single-Pass Adaptive Vector Quantization

CORNEL CONSTANTINESCU AND JAMES A. STORER

Invited Paper

Constantinescu and Storer [4], [5] present a new single-pass adaptive vector quantization algorithm that learns a codebook of variable size and shape entries; they present experiments on a set of test images showing that with no training or prior knowledge of the data, for a given fidelity, the compression achieved typically equals or exceeds that of the JPEG standard. This paper presents improvements in speed (by employing K-D trees), simplicity of codebook entries, and visual quality with no loss in either the amount of compression or the SNR as compared to the original full-search version.

I. INTRODUCTION

Vector quantization is a powerful approach for lossy image compression when a good codebook is supplied, but the need to have this codebook supplied in advance can be a significant drawback. Constantinescu and Storer [4], [5] show how to combine the ability of lossless adaptive dictionary methods to process data in a single pass with the ability of vector quantization accurately to approximate data. For a given overall fidelity of the decompressed image, the compression achieved by this new approach typically equals or exceeds the JPEG standard. In addition, it often outperforms traditional trained VQ (even in the best case, where the codebook is specifically trained for the type of data being compressed) while at the same time having a number of additional advantages: First, it is a single-pass adaptive algorithm (requiring no codebook to be provided in advance). Second, one can provide precise guarantees in advance on the distortion of any $l \times l$ subblock of the image (whereas trained VQ simply finds the best match to an available codebook). Third, with a fixed codebook size, one can continuously vary the fidelity/compression tradeoff (whereas trained VQ typically achieves different tradeoffs by employing multiple codebooks). Our algorithm also enjoys some of the advantages of trained VQ, such as fast table-lookup decoding.

Manuscript received November 1, 1993; revised January 15, 1994.

The authors are with the Department of Computer Science, Brandeis University, Waltham, MA 02254 USA.

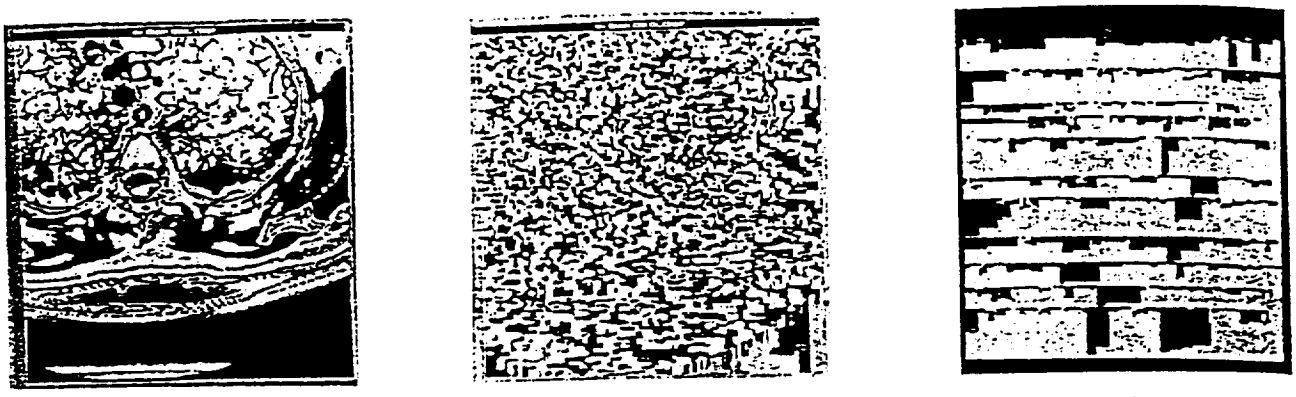
IEEE Log Number 9401248.

This paper presents improvements in speed, simplicity of codebook entries, and visual quality with no loss in either the amount of compression or the signal-to-noise ratio (SNR) as compared to the original full-search version. Section II reviews the basic single-pass adaptive VQ algorithm presented in Constantinescu and Storer [4], [5]. Section III presents a k–d tree implementation of the dictionary that greatly improves the speed of serial implementations with no loss in either the amount of compression or the SNR as compared to the original full search version. In fact, due to a minor improvement in the basic algorithm (see the end of Section II), the experiments reported here improve upon what is reported in Constantinescu and Storer [4], [5]. Section IV presents a new learning heuristic that employs only square-shaped entries. Section V presents a new method for distortion computation that improves visual quality without any significant sacrifice in the SNR. Section VI mentions some current areas of research.

II. THE BASIC SINGLE-PASS ADAPTIVE VQ ALGORITHM

In this section we review the work presented in [4], [5]. As mentioned in the Introduction, one can view this approach as combining ideas from adaptive lossless compression and from vector quantization.

With lossless *adaptive dictionary methods*, a local dictionary D is used to store a constantly changing set of strings. Data are compressed by replacing substrings of the input stream that also occur in D by the corresponding index into D; we refer to such indices as *pointers*. The encoding and decoding algorithms work in lockstep to maintain identical copies of D (which is constantly changing). The encoder uses a *match heuristic* to find a match between the incoming characters of the input stream and the dictionary, removes these characters from the input stream, transmits the index of the corresponding dictionary entry, and updates the dictionary with an *update heuristic* that depends on the current contents of the dictionary and the match that was just found. If there is not enough room left in the dictionary, a *deletion heuristic* is used to delete an existing entry. For



(a) (b) (c)

Fig. 1. (a) ChestCAT original. (b) ChestCAT map. (c) ChestCAT dictionary.

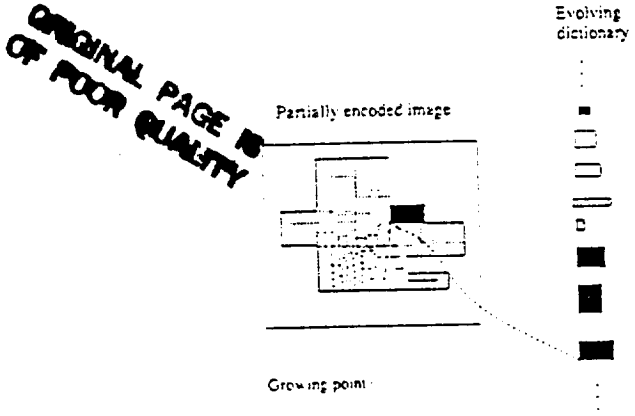


Fig. 2. On-line adaptive VQ.

an overview on adaptive lossless compression, see the book by Storer [14].

Vector quantization is a lossy method that compresses an image by replacing subblocks by indices into a dictionary of subblocks. Traditionally, the subblocks are all the same size and shape and the dictionary must be computed in advance by "training" on sample data. Not only can training be computationally expensive, but "full-search" encoding that is guaranteed to find the closest vector in the dictionary can also be very time-consuming. In practice, tree-structured dictionaries are often used. Lin [10] studies the performance—complexity tradeoffs for vector quantization. See Gersho and Gray [9] for an introduction to vector quantization and references to the literature.

The basic single-pass adaptive VQ algorithm presented in [4], [5] is depicted in Fig. 2, which is followed by Algorithms 1a and 1b, the Lossy Generic Encoding and Decoding Algorithms for on-line adaptive vector quantization. Fig. 1 illustrates the algorithms by showing for a CAT-scan chest image (Fig. 1(a)), a map of how the compressor covers the image with rectangles (Fig. 1(b)), and a portion of the dictionary (Fig. 1(c)) about halfway through the compression process. The operation of the generic algorithms is guided by the following heuristics:

The Growing Heuristic: The heuristic selects one growing point GP(x, y, q) from the available pool GPP. All

1) Initialize the *local dictionary* D to have one entry for each pixel of the input alphabet and the *growing points pool (GPP)* with one (or more) growing points.
2) Repeat until there are no more growing points in *GPP*:
   a) {*Select the next growing point from GPP:*}
      Use a *growing heuristic* to choose a growing point GP from *GPP*.
   b) {*Get the best match block* b :}
      Use a *match heuristic* to find a block b : in D that matches with acceptable fidelity *image*$(GP, b:)$ (the portion of image determined by GP having the same size as b). Transmit $\lceil \log_2 |D| \rceil$ bits for the index of b :.
   c) {*Update* D *and GPP:*}
      Add each of the blocks specified by a *dictionary update heuristic* to D (if D is full, first use a *deletion heuristic* to make space)

Algorithm 1a: Lossy Generic Encoding Algorithm.

1) {Initialize D and *GPP* by performing Step 1) of the encoding algorithm.}
2) Repeat until there are no more growing points in *GPP*:
   a) {Select the next growing point from *GPP*:}
      Perform Step 2a of the encoding algorithm to obtain GP.
   b) {Get the best match block b:}
      Receive $\lceil \log_2 |D| \rceil$ bits for the index b. Retrieve b from D and output b at the position determined by GP.
   c) {Update D and *GPP*:}
      Perform Step 2c of the encoding algorithm

Algorithm 1b: Lossy Generic Decoding Algorithm.

experiments reported here use the *wave* heuristic (a "wave front" that goes from the upper left corner down to the lower right corner). Other examples of growing heuristics include *circular* (a "ball" that expands outward from the center), *diagonal* (a successive "thickening" of the main diagonal), and FIFO (first-in first-out).

The Match Heuristic: This heuristic decides what block b from the dictionary D best matches *image*GP (the portion of the image of the same shape as b defined by the currently selected growing point GP). All experimental results reported here use the *greedy* heuristic (choose the largest match possible of acceptable quality, and among two matches of equal size, choose the one of best quality). The parameters that guide the matching process are: The *distance measure*; we use the standard mean-square measure in all experiments. The *elementary subblock size* l; large matches can be divided into subblocks of constant

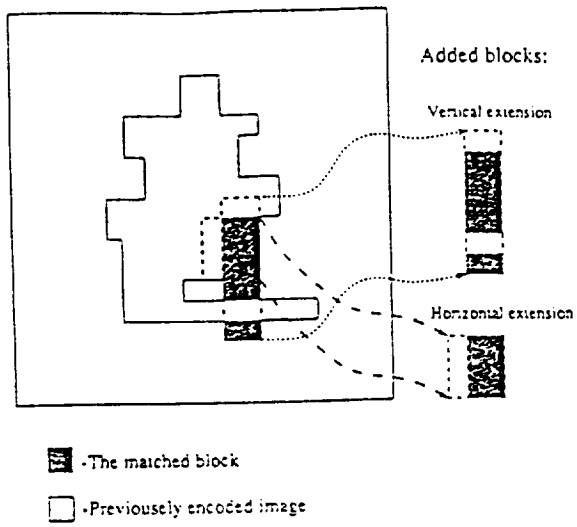


Fig. 3. "OneRow + OneColumn" learning heuristic.

size $l \times l$, and then distance is computed as the maximum distance among the subblocks: this prevents distortion from being unacceptable in a small portion of a match because it is better than needed in other areas (all experiments reported here use $l \times l = 4 \times 4$). The *type of coverage*: examples of image covering strategies include *first* coverage where the distance is computed only on the uncovered part of *image*GP, *last* coverage where the match is computed for the entire block (except if it falls outside the image borders), and *average* coverage (used in all experiments reported here) where the match is computed for the entire block as for last, but on overlapped areas the resulting value is the average value between all the values of matches that happen to cover that pixel. The *threshold* t: a real number that defines the maximum allowed distance (distortion) between *image*GP and b.

The Growing Points Update Heuristic: The growing points update heuristic is responsible for generating new growing points after each new match is made. For all experiments reported here, the concave corners of the partially encoded/decoded image are chosen.

The Dictionary Update Heuristic: The dictionary update heuristic adapts the contents of the dictionary D to the part of the image that is currently encoded/decoded. All experiments reported here use the *OneRow + OneColumn* dictionary update heuristic, depicted in Fig. 3, that adds (if possible) two new blocks to the dictionary, constructed by extending the previously matched block (or part of it) vertically and horizontally by one row.

The Deletion Heuristic: This heuristic maintains the dictionary D so it can have a predefined (constant) size D_{max}. All experiments reported here use the *LRU* heuristic (delete the entry that has been least recently used).

Before closing this section, we should report an experimental finding made after the writing of Constantinescu and Storer [4]. Although experiments have shown that the basic algorithm is robust over a wide choice of heuristics, allowing growth in only one quadrant (as long as possible) typically improves compression (by about 10% on average) for the same SNR. Because *wave* growing can "fill" the entire image and still satisfy the above restriction, this paper has switched from *circular* (used in Constantinescu and Storer [4]) to *wave*.

III. $K-D$ Tree Dictionary Data Structure

The basic algorithm presented in Constantinescu and Storer [4], [5] encodes with simple linear search to find matches, and is very slow if implemented on a standard serial architecture (decompression is essentially table-lookup, and is quite fast). In this section we present a new algorithm based on k–d trees that reduces the search time from minutes or even hours to a few seconds on a UNIX workstation.

If we consider each dictionary block b with $k_b = m_b \times n_b$ pixels as a point in a k_b-dimensional space, the problem is to find the closest point (best block) to a given point (image area *image*GP) from a set of points (dictionary of blocks); that is, a nearest neighbor search problem (e.g., Preparata [13], Dasarathy [6]). However, the problem has several nontrivial peculiarities: First, the dictionary blocks have *variable dimension* (k_b) and *variable shape* (m_b and n_b can have arbitrary values). Second, the dictionary maintains a *dynamic* set of blocks: in addition to search we need *insertions* and *deletions*. And third, the "best" block is defined by a match heuristic that may use a variety of distortion measures that work over a variety of rectangle sizes (and there is always a perfect match to the unit size). Typically, nearest neighbor algorithms perform time-consuming preprocessing in order to have fast processing time. This works well if the set of points is *static* (does not change during processing). However, in our case the set of points (dictionary) consists of the alphabet at the beginning of encoding, and changes during encoding, on average with two insertions and eventually two deletions for each search.

We have employed a data structure based on k–d trees (e.g., Bentley [1], Bentley and Friedman [2], Overmars and van Leeuven [12]). Each branch in the tree relies on some *discriminating dimension* and a *partition value*. The nonterminal nodes contains the (two) pointers to the sons, the partition value, and the discriminating dimension (which can be data-dependent); terminal nodes (named *buckets*) contains data (dictionary blocks). Because we are using the wave growing heuristic, we can assume that a region that is being matched is always "attached" to the already compressed portion of the image at its upper left corner, and we use the upper left 4×4 subblock of the region to provide the keys for the search. To find matches that are less than 4 pixels in either dimension, we employ a few additional trees, as to be discussed shortly.

A significant difference between our algorithm and Friedman, Bentley, and Finkel [7] algorithm is that we have a bound on the allowable distortion (the distortion threshold d) before starting the search. So, we can start a range search for the "best" block using the distortion threshold to define the range (instead of going first for some nearest neighbor block, compute the distance r between this block and the query block, and then do a range search backward—the

ChestCAT: Cat-scan chest image, 512 by 512 pixels, 8 bits per pixel.

BrainMrSide: Magnetic resonance medical image that shows a side cross-section of a head, 256 by 256 pixels, 8 bits per pixel; this is the medical image used by Gray, Cosman, and Riskin [GCR91].

BrainMrTop: Magnetic resonance medical image that shows a top cross-section of a head, 256 by 256 pixels, 8 bits per pixel.

NASA5: Band 5 of a 7-band image of Donaldsonville, LA; the least compressible of the 7 bands by UNIX compress.

NASA6: Band 6 of a 7-band image of Donaldsonville, LA; the most compressible of the 7 bands by UNIX compress.

WomanHat: The standard woman in the hat photo, 512 by 512 pixels, 8 bits per pixel.

LivingRoom: Two people in the living room of an old house with light coming in the window, 512 by 512 pixels, 8 bits per pixel.

FingerPrint: An FBI finger print image, 768 by 768 pixels, 8 bits per pixel; includes some text at the top.

HandWriting: The first two paragraphs and part of the figure of page 165 of *Image and Text Compression* (Kluwer Academic Press, Norwell, MA) written by hand on a 10 inch high by 7.5 inch wide piece of gray stationary scanned at 128 pixels per inch, 8 bits per pixel; approximately 1.2 million bytes.

Fig. 4. Description of the images.

so-called "bounds-overlap-ball" test). If we use the range $[x_i - d, x_i + d]$ for each dimension i of the query block x (key area), deciding to go left, right, or both ways in the k–d tree depending on how this range compares with the partition value v_i associated with the currently visited nonterminal node, we end up by selecting *all* potential best matches (all blocks which meet the distortion threshold on the key area), no matter what distortion measure we use as long as it is monotonic in dimension values as well as in the number of dimensions (conditions required also by Friedman, Bentley, and Finkel algorithm). An example of such a measure is the standard $L2$ (Euclidean) metric. Although mean-square error does not satisfy this condition, it is a bit faster to compute (because there is no square root to compute) and works equally well in practice.

Let us now consider the complexity of our algorithm when the k–d tree data structure is employed. Encoding time is bounded by

$$O\left(N + \frac{N(S(D_{\max}, m) + Q(N) + m)}{r}\right)$$

where N is the number of pixels in the image, $S(D_{\max}, m)$ is the maximum time to search a dictionary with a maximum of $D_{\max}$ entries each with at most m pixels, $Q(N)$ is the time to insert and delete for the growing points queue, and r is the amount of compression (original size/compressed size). Straightforward implementation of the growing heuristics we have considered uses $O(\log(N))$ time by employing a heap data structure; however, this time can be reduced to $O(1)$ by implementing all heuristics in a manner similar to FIFO. Under ideal assumptions, it can be shown that the expected time for range search in k–d trees is $O(\log n + B)$, where B is the number of blocks found (Bentley and Stanat [3], Friedman, Bentley, and Finkel [7]). If we take $S(D_{\max}, m)$ to be $O(\log(D_{\max}))$ (which from our experiments appears to be a reasonable assumption), the improved encoding time is

$$O\left(N + \frac{N\log(D_{\max})}{r}\right)$$

under the reasonable assumption that $m = O(\log(D_{\max}))$. In many applications, it may be reasonable to assume that r is $\log(D_{\max})$, which brings the encoding time down to $O(N)$ time. As before, decoding is essentially table lookup, and can be done in $O(N)$ time.

Some parameters of the k–d tree should be adjusted by experimentation with real data or simulation because they reflect some compromise between time, memory space, and retrieval quality that is generally dependent on the application domain. After experimenting with a number of alternatives we choose the following settings (used for all the experiments reported in this paper):

Bucket Size: Maximum 8 blocks per bucket. (We experimented with bucket sizes ranging from 1 to 32.)

Discriminating Dimension: The dimension with the largest spread of values (computed by estimating the variance on every dimension of the key, for the 8 blocks in the bucket). (We experimented with random choice, and with cyclic choice depending on the level in the tree).

Partition Value: The mean value between all of the discriminating dimension values in the bucket. (We experimented with random values which worked relatively well).

Range: $1.25 * d$. (Even though mean-square error does not satisfy the monotone properties discussed earlier, by extending the range just a little to $[x_i - 1.25 * d, x_i + 1.25 * d]$, the retrieval quality is as good as for full search with an insignificant increase in search time.)

Number of k–d Trees: Four trees $t1, t2, t3$, and $t4$, with the following key sizes and block assignment:

- t_1 has 1×1 key and contains blocks of size $1 \times n$ or $n \times 1$, with $n \geq 2$.
  (t_1 is simply a binary search tree).
- t_2 has 2×2 key and contains blocks of size $2 \times n$ or $n \times 2$, with $n \geq 2$.
- t_3 has 3×3 key and contains blocks of size $3 \times n$ or $n \times 3$, with $n \geq 3$ and
- t_4 has 4×4 key and contains blocks of size $m \times n$, with $m, n \geq 4$.

Regarding the number of trees to use and the key sizes, since our algorithm is "normalized" by using $l \times l$ elementary areas ($l = 4$ for all experiments reported here), then using a key of size at least $l \times l$, no matter how "good" a big block is on the rest, if it does not satisfy the distortion threshold on the key area it will be rejected also by the full search. Practically, the improvement in selectivity by using keys bigger than 4×4 does not justify the increase in the

Table 1 Comparison of Compression Ratios for the Same SNR (PSNR) (Each of these columns shows the same SNR (the corresponding PSNR is shown in parentheses) the compression achieved by our algorithm with the new tree search data structure, our basic full-search algorithm, and by JPEG.

	"Very Good"		"Good"		"Fair"	
	Quality SNR (PSNR)	Compression TREE/FULL/JPEG	Quality SNR (PSNR)	Compression TREE/FULL/JPEG	Quality SNR (PSNR)	Compression TREE/FULL/JPEG
ChestCAT	29 (36)	4.3/4.3/3.0	22 (29)	8.9/8.9/4.8	18 (25)	12.8/12.7/6.7
BrainMR_Side	28.5 (39)	4.1/4.1/4.6	26.5 (37)	4.8/4.9/6.1	20.5 (31)	10.4/10.3/15.8
BrainMR_Top	27 (35)	2.8/2.9/2.4	20.5 (28.5)	5.7/5.7/3.9	15.5 (23.5)	10.4/10.8/6.6
NASA5	30.5 (41)	4.1/4.2/4.1	28 (38.5)	5.6/5.6/5.9	26 (36.5)	7.4/7.5/8.5
NASA6	46 (51.5)	22.6/22.8/8.4	40.5 (46.5)	74.4/80.1/64.7	39 (45)	107.8/106.5/65.1
WomnHAT	35 (40.5)	4.0/4.1/4.4	30 (35)	8.6/8.8/13.7	27 (32.5)	14.4/14.5/23.5
LivingRoom	32 (38)	3.9/4.0/4.3	27 (33)	7.4/7.5/9.1	24.5 (30.5)	10.8/11.0/14.3
FingerPrint	32 (35)	6.2/6.3/6.5	24 (27)	26.5/26.5/27.3	22 (25)	37.6/38.9/35.0
HandWriting	32 (33)	17.3/17.0/9.5	24.5 (25.5)	61.0/60.1/32.0	17.5 (18.5)	172.0/177.0/67.3

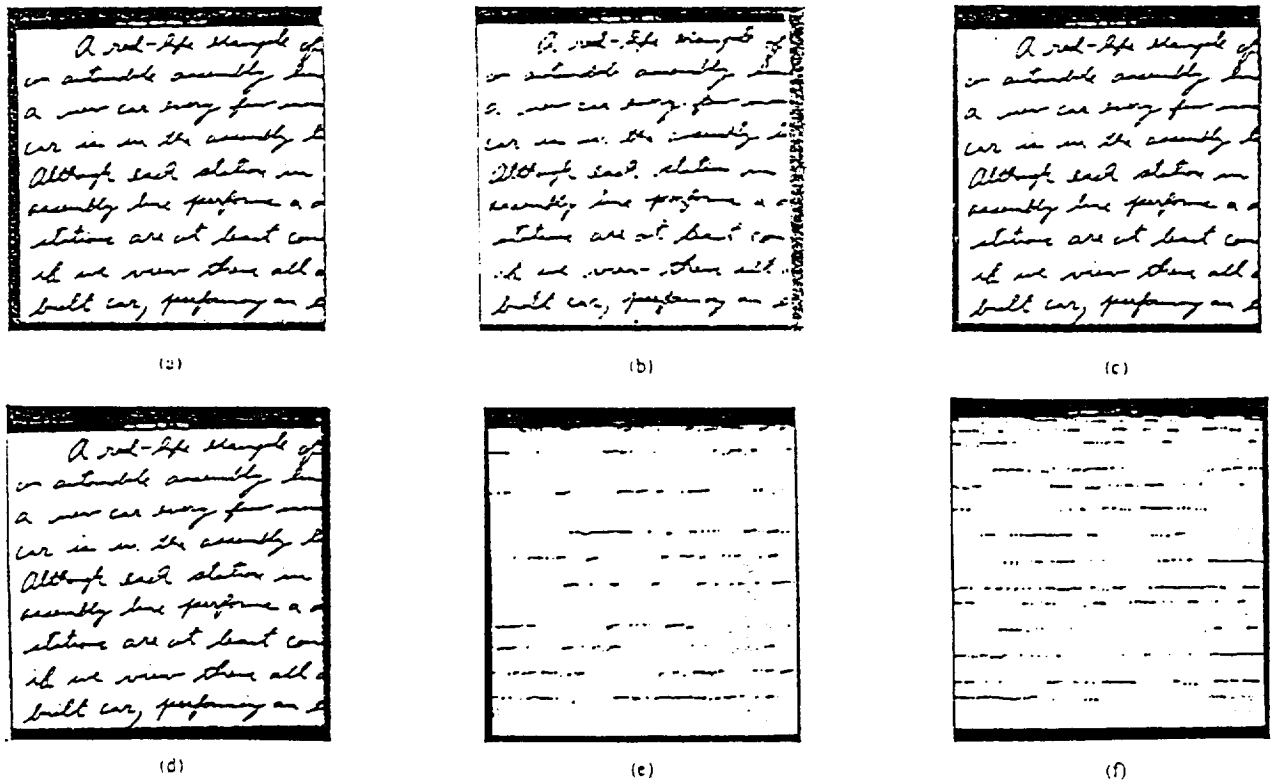


Fig. 5. (a) HandWriting original. (b)HandWriting JPEG at 70-to-1. (c) HandWriting at 70-to-1 using rectangles. (d) HandWriting at 70-to-1 using squares. (e) The dictionary for Fig. 5(c). (f) The dictionary for Fig. 5(d).

tree search time. We experimented with different strategies of searching the forest of 4 trees but no one proved to be significantly better than searching in the order: *t4, t3, t2, t1* where the search goes from one tree to the other *only if* no block was found.

To evaluate the performance of our algorithm, we used the test images described in Fig. 4. For each test image, we adjusted the threshold to get three compressed files, one of very good quality, one of good quality, and one of fair quality; the results are shown in Table 1. Although the compression obtained is nearly identical with the basic full search algorithm, the execution time for a 4 K dictionary was about 60 times faster (roughly speaking, we now use seconds rather than minutes to encode on a UNIX

the "activity" in a region of the image as the ratio between the variance (to the mean) V and the mean M on this region. From experimentation, we can say that if the ratio A is smaller than 4%–5%, then the area is smooth and we use a smaller distortion threshold of $0.4*d$ for this area; if $5\% < A \leq 10\%$ we use an intermediary threshold of $0.6*d$, and if $A > 10\%$ than the area is active and we use the entire threshold d. Figure 6(a) shows our algorithm on the WomanHat image, using a constant distortion threshold at 10-to-1 compression. Figure 6(b) shows the results of the method described above at 10-to-1 compression. For comparison, Fig. 6(c) shows JPEG at 10-to-1 compression. Similarly, Fig. 7(a)–(c) shows the ChestCAT image using constant distortion threshold at 10-to-1 compression, the method described above at 10-to-1 compression, and JPEG at 10-to-1 compression. In both Figs 6(b) and 7(b), the visual quality is much improved (especially on smooth areas such as the shoulder in the WomanHat image and the smooth part with the "X" in the ChestCAT image). By comparison, note that in Fig. 7(c) JPEG is blocky and the edges are not preserved; however, for WomanHat, Fig. 6(b) and (c) has similar visual quality.

VI. Current Research

We are currently working on a number of extensions to the basic approach presented in this paper. First we are continuing experiments to better understand how different heuristics affect performance in terms of both speed and quality. Second, parallel algorithms that run in nearly $O(\sqrt{N})$ time with $O(\sqrt{N})$ processors are possible. Third, of interest are formal proofs addressing compression–fidelity tradeoffs.

References

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, 1975.

[2] J. L. Bentley and J. H. Friedman, "Data structures for range searching," *ACM Comput. Surv.*, vol. 11, no. 4, pp. 397–409, 1979.

[3] J. L. Bentley and D. F. Stanat, "Analysis of range searching in quad trees," *Informat. Process. Lett.*, vol. 3, no. 6, pp. 170–173, 1975.

[4] C. Constantinescu and J. A. Storer, "On-line adaptive vector quantization with variable size codebook entries," in *Proc. IEEE Data Compression Conf.* (Snowbird, UT, 1993). IEEE Computer Soc. Press, pp. 32–41.

[5] ——, "On-line adaptive vector quantization with variable size codebook entries," *J. Informat. Process. Manag.*, to appear, 1994.

[6] B. V. Dasarathy, Ed., *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques.* IEEE Computer Soc. Press, 1991.

[7] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.*, vol. 3, no. 3, pp. 209–226, 1977.

[8] R. M. Gray, P. C. Cosman, and E. A. Riskin, "Combining vector quantization and histogram equalization," in *Proc. IEEE Data Compression Conf.* (Snowbird, UT, 1991). IEEE Computer Soc. Press, pp. 113–118.

[9] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression.* Norwell, MA: Kluwer, 1991.

[10] J. Lin, "Vector quantization for image compression: Algorithms and performance" Ph.D. dissertation, Computer Science Dep., Brandeis University, Waltham, MA, 1992.

[11] C. N. Manikopoulos and H. Sun, "Activity index threshold classification in adaptive vector quantization," in *Conf. Proc. 1988 Int. Conf. on Acoustics, Speech, and Signal Processing* (IEEE), pp. 1235–1239, 1988.

[12] M. H. Overmars and J. van Leeuven, "Dynamic multi-dimensional data structures based on quad- and K-D trees," *Acta Informatica*, vol. 17, pp. 267–285, 1982.

[13] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction.* New York: Springer-Verlag, 1985.

[14] J. A. Storer, *Data Compression: Methods and Theory.* Rockville, MD: Computer Sci. Press, 1988.

Cornel Constantinescu received the M.S. Diploma in computer engineering in 1976 from the Polytechnic University of Bucharest, Bucharest, Rumania.

From 1976 to 1980 he was a software engineer at the State Computer Factory and taught courses in design and use of the Computer Factory products to help in the export of these products (for example, to China). From 1980 to 1990 he was an Assistant Professor in the Computer Science Department at the Polytechnic University of Bucharest. He is presently concluding his work towards the Ph.D. degree at the Computer Science Department, Brandeis University, Waltham, MA.

James A. Storer received the B.A. degree in mathematics and computer science from Cornell University, Ithaca, NY, in 1975, the M.A. degree in computer science from Princeton University, Princeton, NJ, in 1977, and the Ph.D. degree in computer science, also from Princeton University, in 1979.

From 1979 to 1981 he was a researcher at Bell Laboratories, Murray Hill, NJ. In 1981 he accepted an appointment at Brandeis University, Waltham, MA, where he is currently Professor of Computer Science and a member of the Brandeis Center for Complex Systems. His research interests include data compression; text, image, and video processing; parallel computation; computational geometry; VLSI design and layout; machine learning; and algorithm design.

Dr. Storer is a member of the ACM and the IEEE Computer Society.

Split-Merge Video Displacement Estimation

BRUNO CARPENTIERI AND JAMES A. STORER

Invited Paper

Motion Compensation is one of the most effective techniques used in interframe data compression. In this paper we present a parallel block-matching algorithm for estimating interframe displacement of blocks with minimum error. The algorithm is designed for a simple parallel architecture to process video in real time. The blocks may have variable size and shape depending on a split-and-merge technique. The algorithm performs a segmentation of the image into regions (objects) moving in the same direction and uses this knowledge to improve the transmission of the displacement vectors. This segmentation identifies the part of the frame "active" with respect to the previous frame and preserves some of the spatial correlation between blocks.

I. Introduction

Data Compression is essential for the storage and transmission of digital video, where large amounts of data must be handled by devices with a limited bandwidth. For example, digital High Definition Television (HDTV) requires more than 1 billion bits per second in uncompressed form. Knowledge of motion or displacement of groups of pixels in successive frames can be the basis of video compression algorithms and can be used in addition to other classical single-image compression techniques, such as transform, interpolation, and quantization algorithms, to greatly reduce the amount of data transmitted. Here we will restrict our attention to the translational component of the motion and refer to the algorithms that compute the trajectory information of a pixel or a block of pixels as *displacement estimation algorithms*.

Block-Matching Displacement Estimation Algorithms divide a frame into a number of rectangular blocks and compute a displacement vector for each block by correlating the block with a search area in the previous frame; see Jain and Jain [4], Koga *et al.* [7], Srinivasan and Rao [9].

In this paper we present a real-time parallel algorithm for displacement estimation using a two-dimensional grid architecture and then show how the algorithm can be implemented on a pipe. The algorithm is based on a block-matching approach to the problem and uses a split-and-merge technique: the blocks (*superblocks*) have a variable size that is determined at each step of the algorithm from the previous step and the input data. In fact, the algorithm performs a segmentation of the image into areas moving in the same direction and uses this knowledge to improve the transmission of the displacement vectors of the elementary blocks.

In the next section we outline the sequential fixed-size block displacement estimation algorithm presented in Jain and Jain [4]. In Section III we present our new algorithm, Section IV is devoted to its analysis. Section V discusses experimental results, Section VI outlines how the segmentation operated by the Split-Merge technique can be the basis of a full video coder. In Section VII we present our conclusions.

Manuscript received November 1, 1993; revised January 15, 1994.

B. Carpentieri is with Dipartimento di Informatica ed Applicazioni, Università di Salerno, 84081 Baronissi (SA), Italy.

J. A. Storer is with the Computer Science Department, Brandeis University, Waltham, MA 0225 USA

IEEE Log Number 9401249..

II. Image Coding and Displacement Estimation

In this section we review the fixed-size block displacement estimation algorithm proposed by Jain and Jain [4]. This algorithm and its assumptions have been a guideline for more recent work in the field, similar approaches are taken by Koga *et al.* [7], Srinivasan and Rao [9], Kappagantula and Rao [5], Puri *et al.* [8], and Ghanbari [3].

In a typical displacement estimated image coding algorithm the frame is segmented into blocks. For each block a displacement vector is computed and sent to the decoder; moreover, the encoder computes the difference between the the original frame and the frame that the decoder could reconstruct from the displacement vectors, and sends this difference image to the decoder. All data sent from the encoder to the decoder may eventually go through an additional entropy coding phase.

A. Displacement Estimation

The algorithm proposed in Jain and Jain [4] segments an image into fixed-size small rectangular blocks, each block assumed to be undergoing independent translation.

If these areas are small enough, rotation, zooming, etc., of larger objects can be closely approximated by piecewise translation of these smaller areas. The goal is to approximate interframe motion by piecewise translation of one or more areas of a frame relative to a reference frame. Let U be an $M \times N$ size block of an image and U_r be an $(M+2p) \times (N+2p)$ size area of a reference (neighboring) image, centered at the same spatial location as U, where p is the maximum displacement allowed in either direction in integer number of pixels. The algorithm requires for each block a search of the *direction of minimum distortion* (DMD), i.e., of the displacement vector that minimizes a given distortion function. A possible mean distortion function between U and U_r is defined in Jain and Jain [4] as

$$D(i,j) = \frac{1}{MN} \sum_{m=1}^{M} \sum_{n=1}^{N} g(u(m,n) - u_r(m+i, n+j)), \qquad -p \le i,j \le p$$

where $g(x)$ is a given positive and increasing distortion function of x. The direction of minimum distortion is given by (i,j), such that $D(i,j)$ is minimum.

One problem of this approach is that finding optimal displacements requires the evaluation of $D(i,j)$ for $(2p+1) \times (2p+1)$ directions per block. For example, even for motions up to 5 pixels along either side of the axes a search of 121 positions per block is required. The solution proposed in Jain and Jain [4] is to assume that the data are such that the distortion function monotonically increases as we move away from the DMD along any direction in each of the four quadrants. This assumption makes possible a search procedure for the DMD that is an extension in two dimensions of the standard logarithmic search in one dimension (see Knuth [6]).

In the next section we present a parallel algorithm that eliminates the need for this assumption and which can be implemented to run on-line on a practical parallel architecture.

III. A Split-Merge Parallel Block-Matching Algorithm

In this section we present a new parallel block-matching algorithm for displacement estimation based on a split-and-merge technique taking advantage of the fact that groups of blocks often move in the same direction (for instance, if they are part of the same object or part of the background). The encoding algorithm computes the displacement vectors (in parallel) and sends them in compact form to the decoder. The decoder receives the data and constructs an approximate version of the image, which will be corrected in the next step of the general encoding algorithm.

A. The Model of Computation

To process frames of n pixels each, the encoding algorithm employs a $\sqrt{N} \times \sqrt{N}$ grid of processors, $1 \le N \le n$, each having $O(n/N)$ local memory. Although all of what we present is well defined when $N \ll n$, to simplify our presentation we shall assume $N = kn$ for some $0 < k \le 1$ (and here each processor has $O(1)$ local memory). For decoding we will need only a single processor with $O(n)$ memory.

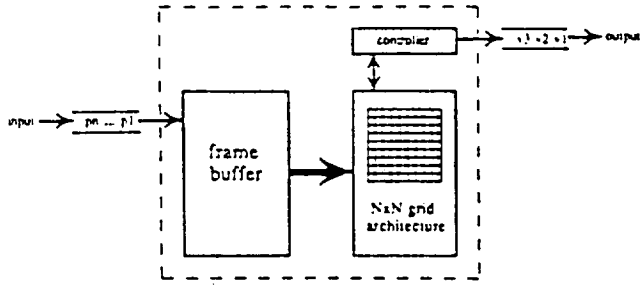


Fig. 1. Displacement estimation encoder.

Each frame is divided into N rectangular blocks numbered in the same way as the processors; we assume that at time t processor i receives as input block i from the tth frame. Since each processor corresponds to a block, and *vice versa*, from now on we will use the terms processor and block interchangeably.

The encoding algorithm implies the use of a sequential *controller* to monitor the execution of the algorithm. The controller will need $O(N)$ dynamic memory and will perform communication operations only with processor 1. We will identify this controller with processor 1 itself by allocating to this processor an additional $O(N)$ local dynamic memory. The encoder computes the displacement vectors and transmits them in a compact form to the decoder on a serial line. Figure 1 depicts our model of computation. The input frames come to the frame buffer on a high-speed communication line, in time proportional to n. The data flow from the frame buffer to the grid architecture that performs the search of the optimal displacement for each block. The communication between the frame buffer and the grid architecture has to be performed fast enough to allow the grid time to perform the necessary computation on the actual frame before receiving the next frame. In fact, the bold arrow implies that this communication should be performed either in parallel or on a serial line with a speed of cn/N pixels per unity of time, where c is a system-dependent constant. In Fig. 2 is shown a possible implementation of the frame buffer: embedded into the grid. The input is pipelined through the processors. At each step each processor can pass the input to its neighbor and, when necessary, can simultaneously copy it into its own working memory.

B. The Encoder Algorithm

Figure 3 shows the encoder algorithm at time t. Each processor at time t computes in parallel the displacement of the block that it represents (in frame t) with respect to a search area in frame $t-1$. For simplicity we assume that the size of the search area is exactly 3×3 blocks, that is, for each processor we limit the search area to its adjacent blocks. Processor i at time t keeps the description of the block it represented at time $t-1$ in the variable $\text{block}_{pf}(i)$

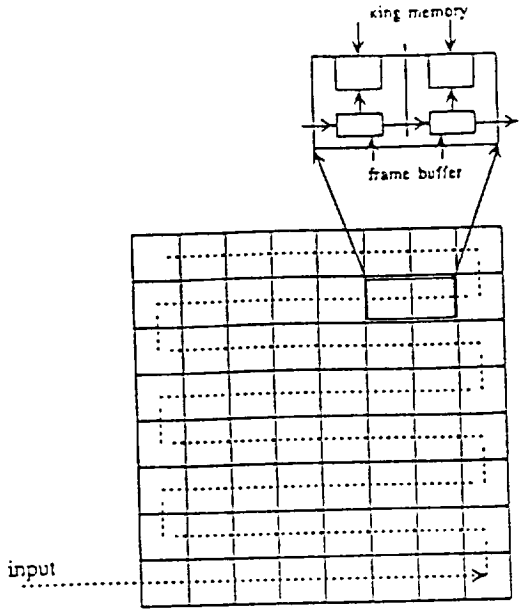


Fig. 2. A frame buffer implementation.

(the subscript pf is short for "previous frame"). If at time $t-1$ a number of pairwise adjacent blocks have the same displacement vector, then at time t they are considered to be a well-defined *superblock*: *superblock(i)* where i is the leader of the *superblock* (the processor with minimum ID). If they continue to move together in the same direction at time t, just a single displacement vector for the whole superblock is sent from the encoder to the decoder. Each processor is not aware of the shape of the *superblock* to which it belongs, but is aware of the adjacent processors that move in its direction (*coblock*). The union of *coblocks* for adjacent processors that move in the same direction will define a *superblock*. At each time t the algorithm can be divided into three steps. In Phase 1 (*the compute phase*) the displacement vector for each single block is computed and each processor compares its displacement with the displacement of the adjacent processors. Each processor i keeps a list of the adjacent processors that move in its same direction: *coblock(i)*. In Phase 3, whenever this is possible, these lists will be merged together into *superblocks*. At time $t+1$ a single displacement vector will be sent from the encoder to the decoder for all the processors in a *superblock* that still move in the same direction.

In Phase 2 (*the split-and-send phase*) processor 1, the processor in the upper left corner of the grid, becomes the controller and communicates with the others processors: gathering information on their displacement vectors, deciding their belonging to a *superblock* or the occurrence of a split, (i.e., whereby processors leave a *superblock* because their motion differs from that of the majority. We address the complexity of this operation in the next section.). Being aware of all the displacement vectors for the N processors, the controller, for each *superblock* tests if *splits* have occurred and constructs the *list-of-splits*, i.e., the list that specifies which processors that were assumed to be part of a *superblock* are no longer part of it because

```
Phase 1: (COMPUTE)

for each processor i in parallel do:
  begin
  1) for every adjacent processor neighbor
                                  do get block v_pf(neighbor)
  2) COMPUTE (full search) its own displacement vector v_i(i)
  3) for every adjacent processor neighbor do get v_i(neighbor)
  4) set coblock(i) = set of adjacent processors neighbor such
                                  that v_i(neighbor) = v_i(i)
  end

Phase 2: (SPLIT and SEND)

controller do
  begin
  1) for i = 1 to N do
       get v_i(i) and store it into the record representing i in
                          the superblock to which i belongs
  2) for each superblock do
       begin
  2.1) SPLIT the superblock into groups of processors j
                  having the same displacement vector v_i(j)
  2.2) let pmin be the processor with minimum ID in the larger group
  2.3) if pmin is not the leader of the current superblock then make a
       new superblock with leader pmin and displacement vector v_i(pmin)
  2.4) add the other groups to list-of-splits
  2.5) if pmin is not the leader of the current superblock then
                          delete the current superblock
  3) if length (list-of-splits ) > T  then for i =1 to N do SEND v_i(i)
                          else
                          begin
                          SEND list-of-splits
                          for each superblock (i ) do
                                  SEND v_i (superblock (i ))
                          end
  end

Phase 3: (MERGE)

controller do
  begin
  1) for i = 1 to N do get coblock (i )
  2) from the coblocks at time t construct the new superblocks at time t+1
  end
```

Fig. 3. The algorithm at time t.

they are now moving in a different direction with respect to the rest of the *superblock*. If the length of *list-of-splits* is less than a threshold T, then the controller sends the *list-of-splits* and the displacement vectors for the *superblocks*; otherwise, it sends the displacement vectors for each single block. The threshold monitors the efficiency in terms of amount of data sent to the decoder of sending both *list-of-splits* and the *superblocks* displacement vectors, instead of the displacement vectors for each single block.

In Phase 3 the *superblock* s at time $t+1$ are built from the *coblocks*. The encoder and the decoder maintain dynamically a list of the *superblocks*, i.e., of which block belongs to which *superblock*. No communication between the encoder and the decoder is needed to maintain the description of the *superblocks*: the decoder has enough information to compute the shape of the *superblocks* at time $t+1$. At every time t a list of the positions in which a *split* has occurred is sent from the encoder to the decoder: in this way the decoder is able to decode the displacement vectors sent from the encoder.

C. The Decoder Algorithm

The decoder receives at time t the information sent from the encoder during Phase 2. It has computed at time $t-1$ the *superblocks* at time t and therefore it can assign to each block the correspondent displacement vector. Finally, it has enough information to compute which blocks will be in which *superblocks* at time $t+1$. The decoder is not a parallel machine: one single processor suffices to perform

the necessary operations. The decoder uses $O(N)$ memory to decode each frame in $O(N)$ time.

D. Splits and Displacement Vectors

One of the critical points of the algorithm is the communication from the encoder to the decoder of the *list-of-splits*, i.e., of the list of the processors that at time t belonged to a *superblock* but no longer do, and of their displacement vectors. There are two requirements that the *list-of-splits* must satisfy: it must be computationally easy to build, and it must have a concise encoding; otherwise, sending only one displacement vector for each *superblock* would not be convenient because of the necessity of sending also the *list-of-splits*.

The *list-of-splits* is dynamically built: In line 2.4 of Phase 2, *groups* of processors are added to the list, a single displacement vector per each group. We keep a hash table of the possible displacement vectors: each time a group is added to the list we compute the hash value of its displacement vector and we associate to the corresponding entry in the table this displacement vector and the list of the processors in the *group*. This list begins with the ID of the smaller processor, then the ID's of the other processors follow, each coded in terms of the displacement with respect to the previous one. Because the processors were part of the same *superblock* and are still moving in the same direction, we can expect their ID numbers to be very close and we can get good compression with this simple heuristic. When the encoder sends the *list-of-splits*, it sends each nonzero entry in the table.

There might be more than one solution to the computation in Line 2 of Phase 1. The block examined could match optimally more than one block in the search area, or else we may want to consider in the next Phase more than one direction in which the block can move, in such a way to have more options when it is time to shape the *superblocks*. A way to do this is to save for each block all the displacement vectors that allow an error less than a threshold t when the block is matched in the search area. In this case, in line 1 of Phase 2, the processor sends to the controller not only a single vector but a list of possible vectors.

To determine the eventuality of a *split*, in line 2.1 of Phase 2, the controller shall compute in which of the possible directions the majority of the processors move. The number of possible directions is finite and the computation can be limited in advance by limiting the length of each list of possible vectors to an appropriately chosen constant L. Phase 3 is not affected by considering more than one displacement value per vector in Phase 2: a single displacement vector per block has been sent in Phase 2, and now only that vector has to be considered in Phase 3.

E. Implementation on a Pipe

Figure 4 shows how the algorithm can be implemented on a pipe. The inputs to the pipe are the actual frame and the previous frame reconstructed by the decoder. The input

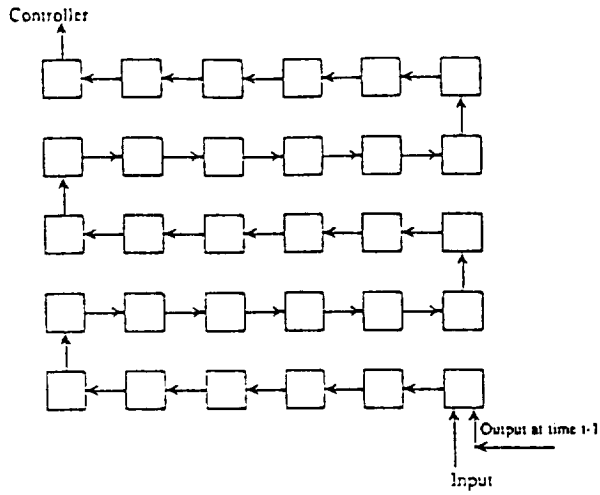


Fig. 4. Implementation of the algorithm on a pipe architecture.

flows in linear time through all the processors. Each processor has to construct the search area by using the information from the previous frame: after $O(N)$ time every processor has available both the block it is representing at the current time and the search area in the previous frame.

The computation involved and the details of the algorithm are analog to the grid implementation.

IV. Analysis of the Algorithm

In this section we analyze the encoder algorithm in terms of complexity, fidelity, and compression. The analysis is done for the grid implementation, similar arguments hold for the pipe implementation.

A. Complexity

Let N be the number of processors in the grid, where $N = kn$ for $0 < k \leq 1$. In Phase 1 lines 1 and 3 involve direct neighbor communication and take constant time. The computation involved in line 2 is the most expensive part of Phase 1, but it still takes constant time, where the constant depends on the size of the search area. The `for` loop in line 1 of Phase 2 might seem to involve $O(N^2)$ communication on a grid architecture: processor 1 has to interact with all the other processors. If we number the blocks by row and column this `for` loop can be easily pipelined as showed in Fig. 5. Therefore, processor 1 will always interact at each iteration of the loop with an adjacent processor: processor 2, and the loop will take $O(N)$ time. The complexity of line 2 (2.1–2.5) depends on the number of processor ID's examined. The *superblocks* are pairwise-disjoint sets; therefore, line 2 has a time complexity of $O(N)$. Line 3 involves also $O(N)$ time.

The `for` loop of line 1 of Phase 3 can be pipelined and takes $O(N)$. For each vector the *coblocks* have a constant size (each processor has at most eight neighbors), therefore, line 2 has time complexity $O(N)$.

In fact, the whole algorithm has at each step t a time complexity $O(N) = O(kn)$, i.e., linear in the size of the input, it is an on-line algorithm.

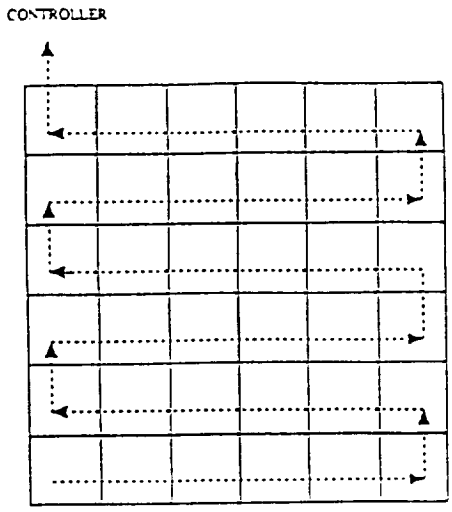


Fig. 5. Pipelining.

Each processor, with the exception of the controller, needs a constant amount of memory. The controller needs $O(N)$ dynamic memory to represent the *superblocks* and to store the *coblocks* and the displacement vectors.

B. Fidelity

The displacement vectors computed by our algorithm are at least as accurate and generally more than those computed by the sequential algorithm: we do not assume any *a priori* hypothesis to simplify the search, rather we search all the possible directions.

C. Compression

The amount of data sent from the encoder to the decoder in our algorithm is in the worst case equal to the amount sent by the fixed-size block algorithm, but the algorithm has the possibility to transmit much less data.

The size of the blocks is chosen in such a way as to approximate different movements of an object by piecewise translation of the blocks themselves. An object may be composed of a large number of blocks, all of which move in the same direction, even in the case when the motion in the sequence is due to a movement of the camera. If neighboring objects move in the same direction with the same speed, they will belong to the same superblock. In fact, a simple but important case is when large groups of pixels comprise "Background" scenery that stays relatively constant from one frame to the next.

The *superblocks* will generally consist of many processors, the length of the *list-of-splits* will be negligible with respect to the cardinality of the *superblocks*, and at each time t a sensible reduction in size of the data sent from the encoder to the decoder is expected. However, when the length of the *list-of-splits* becomes bigger than the threshold T, the controller acts as in the fixed-size algorithm and sends to the decoder one displacement vector per block, starting from $\vec{v}_t(1)$ to $\vec{v}_t(n)$, instead of sending *list-of-splits* and the displacement vectors for the *superblocks*. In this way, it sends the same amount of data that the algorithm in Jain and Jain [4] would have sent. The decoder can infer that the displacement vectors received refer to the blocks, and not to the *superblocks*, from the number of vectors received.

Fig. 6. First and last frame of the sequence "Salesman."

Fig. 7. First and last frame of the sequence "Fog."

Fig. 8. First and last frame of the sequence "Kids."

Fig. 9. First and last frame of the sequence "Mountains."

Fig. 10. First and last frame of the sequence "Pastorale."

V. Experimental Results

We have performed experiments with the following data set (Figs. 6–10 show the first and the last frame of each of these sequences):

Salesman

This sequence is one of the standard test sequences in video compression. It is currently available for anonymous ftp at ipl.rpi.edu and consists of 448 frames, 360 × 288, 8 bits per pixel. It contains relatively little detail or motion,

typical of the head and shoulder sequences common in video-telephone applications.

Fog

From the motion picture "Casablanca," the final scene when Humphrey Bogart and Ingrid Bergman say good-bye in the fog at the airport. This sequence is composed of 60 frames, 152 × 114, 8 bits per pixel, digitized at a rate of 12 frames per second. There is a considerable amount of noisy movement due to the foggy background.

Kids

From the motion picture "It's a Wonderful Life," it is one of the first scenes, where kids (the main characters as children) are sitting at a desk. This sequence is composed of 100 frames, 152 × 114, 8 bits per pixel, digitized at a rate of 12 frames per second. There is a fair amount of movement due to the presence of three characters.

Mountains

From the motion picture "The Sound of Music," one of the final scenes, where the main characters are walking in the mountains. This sequence is composed of 60 frames, 152 × 114, 8 bits per pixel, digitized at a rate of 12 frames per second. The scene involve a noticeable amount of movement.

Pastorale

From the motion picture "Fantasia," a scene from the part of the movie illustrating Beethoven's 6th Syimphony. This sequence is composed of 60 frames, 152 × 114, 8 bits per pixel, digitized at a rate of 12 frames per second.

We define, as usually, the SNR correlation (in decibels), between two frames X and Y, of dimension $M \times N$ as

$$\mathrm{SNR}(X,Y) = 10 \times \log_{10} \frac{\sum_{i<M, j<N} (X(i,j))^2}{\sum_{i<M, j<N} (X(i,j) - Y(i,j))^2}.$$

To describe the amount of movement present in each of the test sequences, Fig. 11 presents for each sequence the SNR correlation between pair of consecutive frames. On the Y axis we plot the SNR correlation, in decibels, between a frame and the previous one, on the X axis the frame number. We can see, for example, that in the sequence "Kids" and in the sequence "Mountains" (Fig. 11(c), (d)) there is at first a higher amount of movement (the first 20 frames of "Kids" and the first 30 of "Mountains"), and then a lower amount of motion. Therefore, the graphs show very low points for the first part of the sequence and then a brisk increase and a smoother behavior. In the sequence "Kids," this is due to the fact that in the first 20 frames the blonde girl moves from the left corner of the picture and sits down at the desk while the boy gets closer, then in the rest of the sequence the two girls and the boy move slightly and chat. In the sequence "Mountains," at the beginning people are

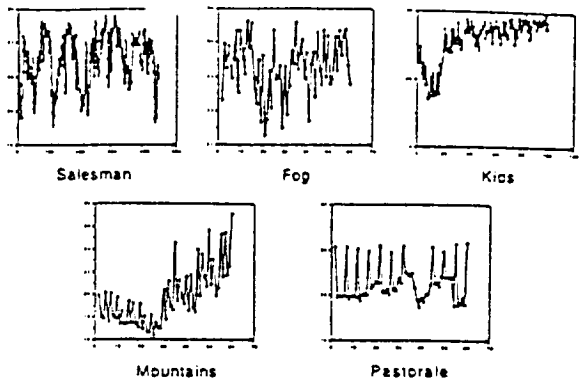


Fig. 11. Motion in the test sequences. X = frame number; Y = SNR (dB) correlation with the previous frame.

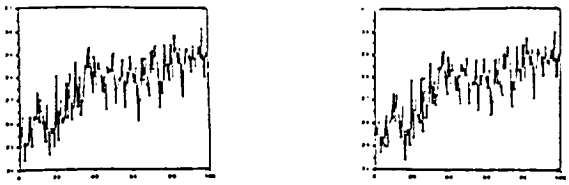
Sequence	Correlation (Previous Frame)	Full Search bs8 vs Split Merge bs4		Full Search bs4 vs Split Merge bs2	
	SNR	SNR	SIZE	SNR	SIZE
Salesman	22.91	25.57 db	18225.5 vs 444.07	26.57 db	5670 vs 3733
Mountains	19.81 db	23.48 db	300 vs 138.96	24.92 db	931 vs 857.03
Fog	34.29 db	35.22 db	300 vs 141.06	36.94 db	931 vs 1248
Kids	25.87 db	27.59 db	300 vs 84.47	28.38 db	931 vs 695.58
Pastorale	22.79 db	24.12 db	300 vs 90.71	27.70 db	931 vs 893.20

Fig. 12. Comparison with the standard full search, fixed-size block, algorithm.

walking fast to the top of the hill but at the end they slow down and turn to the mountains.

Figure 12 shows, in a table, the results we have obtained comparing our algorithm to the standard full search algorithm. The first column of the table identifies the sequence, the second column reports for each sequence the average SNR (in decibels) between consecutive frames as a measure of their correlation. The third and fourth columns present the results of the comparison between the full search algorithm and the Split-Merge algorithm for the test sequences. We have run the full search algorithm with block size 8 (8 pixels by 8 pixels blocks) and block size 4 (4 pixels by 4 pixels blocks) and we have reported in the first subcolumns of the third and fourth columns the average SNR between the original frames and the prediction obtained. Then we have run our algorithm setting the parameters in such a way to achieve that same average SNR and in the second subcolumns we have compared the size of the predictions, i.e., the number of bytes needed to send the prediction from the encoder to the decoder assuming no lossless compression is performed.

As can be seen in Fig. 12, for the same SNR, our algorithm has in general a noticeable saving in size respect to the full search algorithm. In the sequence "Fog" the foggy background produces noisy effects on the segmenta-



(a) (b)

Fig. 13. Sequence "Kids," comparison of Full Search block size 8 (a) and Split Merge initial block size 4 (b). X = frame number, Y = SNR (dB).

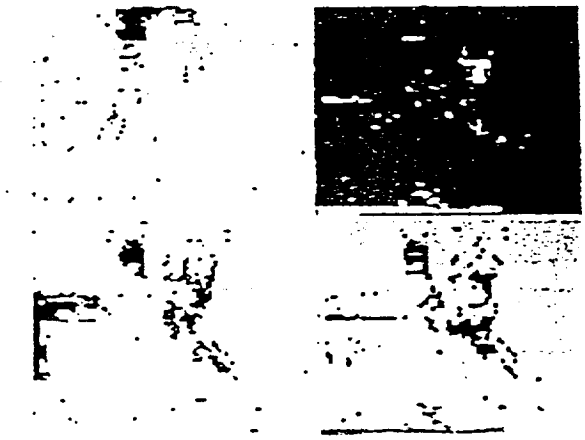


Fig. 14. Segmentation of the frames "Salesman" 100, 200, 300, 447 into superblocks, the initial block size is 4.

tion performed by the Split-Merge algorithm, those effects are particularly relevant when we use a very small initial block size (2 pixels by 2 pixels). This is why the Split-Merge algorithm outperforms the full search algorithm in all experiments but in the case of the sequence "Fog" and initial blocksize 2.

While our analysis has been done in terms of average SNR, it is true that the algorithm performs equally well on a frame-by-frame basis with respect to the full search algorithm. For example, for the sequence "Kids," Fig. 13 shows the SNR values, frame by frame, obtained by the full search algorithm, with block size 8, and the values obtained by the Split-Merge algorithm with initial block size 4 and parameters set to achieve the same average SNR as the full search algorithm. This is true also for all the other sequences tested. On a frame-by-frame basis, the Split-Merge algorithm behaves almost exactly like the full search algorithm.

VI. Splits and Video Coding

This technique suggests a complete video compression algorithm based on the different levels of action that are generally present in a video scene, identified as "splits" and "superblocks." In fact, the segmentation of the frames into superblocks and splits can be the kernel of a complete video compression system. The locations of the splits identify the parts of the frame "active" with respect to the previous frame while the segmentation into superblocks preserves some of the spatial correlation between blocks and avoids some "squaring" effects in the predicted frame.

In Carpentieri and Storer [2] we have presented a video coder based on this Split-Merge displacement estimation technique. The video coder uses the splits and superblocks information to improve the error correction module: two different thresholds are used to determine if a block needs to be corrected, depending on the block being a split or belonging to a superblock: this would not be possible by using the fixed block displacement estimation algorithm which has no notion of spatial correlation between blocks or of active parts of the frame.

Figure 14 shows a segmentation of four frames from the sequence "Salesman," into superblocks; the initial block size is 8. Blocks belonging to the same superblock have the same tone of gray. The splits are depicted by blocks having alternating sequences of black and white pixels.

The splits in Fig. 14 correspond to the parts of the scene that are active in the transition between the previous frame and the actual frame. In fact they are concentrated in the portion of the picture relative to the head of the salesman, to his right hand, and to the object in his hand.

VII. Conclusion

We have presented a new on-line parallel algorithm for displacement estimation based on the block-matching approach, as well as an on-line parallel implementation of this algorithm. At each time t both the decoder and the encoder have available the description of the *superblocks* computed at time $t-1$. Each *superblock* is a set of contiguous blocks that move in the same direction. The partition of the image into *superblocks* corresponds to an approximate segmentation of the image into areas (objects) that move in the same direction. The quality of the approximation depends on the granularity chosen (i.e., the size of the blocks and the setting of the internal parameters). Our algorithm uses this knowledge of the segmentation of the frames to optimize the transmission of the displacement vectors.

Segmenting frames into superblocks preserves the spatial correlation between the blocks in the superblock. This may improve the visual quality of the prediction. Because the splits represent blocks that are in a certain sense "new" with respect to the previous frame, a different degree of correction accuracy can be used for blocks that are splits.

References

[1] B. Carpentieri and J. A. Storer, "A split-merge parallel block-matching algorithm for video displacement estimation," in *Proc. IEEE DCC92* (Snowbird, Mar. 1992).

[2] ——, "Split-merge displacement estimated video compression," in *Proc. 7th Int. Conf. on Image Analysis and Processing* (Bari, Italy, Sept. 1993).

[3] M. Ghanbari, "The cross-search algorithm for motion estimation," *IEEE Trans. Commun.*, vol. 38, no. 7, July 1990.

[4] J. R. Jain and A. K. Jain, "Displacement measurement and its applications in interframe image coding," *IEEE Trans. Commun.*, vol. COM-29, no. 12, pp. 1799–1808, Dec. 1981.

[5] S. Kappagantula and K. R. Rao, Motion compensated predictive coding," in *Proc. Int. Tech. Symp. SPIE* (San Diego, CA, Aug. 1983).
[6] D. E. Knuth, *Searching and Sorting* vol. 3 of*The Art of Computer Programming*. Reading, MA: Addison-Welsey, 1983.
[7] T. Koga, K. Inuma, A. Hirano, Y. Iijima, and T Ishiguro, "Motion-compensated interframe coding for video conferencing," in *NTC 81, Proc.*, Dec. 1981.
[8] A. Puri, H. M. Hang, and D. L. Shilling, "An efficient block matching algorithm for motion compensated coding," in *Proc. ICASSP*, pp. 25.4.1–25.4.4, 1987.
[9] R. Srinivasan and K. R. Rao, "Predictive coding based on efficient motion estimation," in *ICC Proc.*, May 1984.

Bruno Carpentieri received the "Laurea" degree in computer science from the University of Salerno, Italy, in 1988, and the M.A. degree in computer science from Brandeis University, Waltham, MA, in 1990, where he is presently completing the requirements for the Ph.D. degree.

Since 1991 he has been Assistant Professor of Computer Science (Ricercatore) the Dipartimento di Informatica ed Applicazioni at the University of Salerno. His research interests include data compression, in particular video compression and motion estimation, parallel computing, and theory of computation.

James A. Storer received the B.A. degree in mathematics and computer science from Cornell University, Ithaca, NY, in 1975, the M.A. degree in computer science from Princeton University, Princeton, NJ, in 1977, and the Ph.D. degree in computer science, also from Princeton University, in 1979.

From 1979 to 1981 he was a researcher at Bell Laboratories, Murray Hill, NJ. In 1981 he accepted an appointment at Brandeis University, Waltham, MA, where he is currently Professor of Computer Science, chair of the Computer Science Department, and a member of the Brandeis Center for Complex Systems. His research interests include data compression; text, image, and video processing; parallel computation; computational geometry; VLSI design and layout; machine learning; and algorithm design.

Dr. Storer is a member of the ACM and the IEEE Computer Society.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1994	Contractor Report

4. TITLE AND SUBTITLE

High Performance Compression of Science Data

5. FUNDING NUMBERS

930

6. AUTHOR(S)

James Storer and Martin Cohn

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Department
Brandeis University
Waltham, MA 02254-9110

8. PERFORMING ORGANIZATION REPORT NUMBER

NAS5-32337
5555-11
NAS5-31354

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration - HQ/ OSSA
Washington, D.C. 20546-0001

Universities Space Research Association
10227 Wincopin Circle, Suite 212
Columbia, MD 21044

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

CR-189388

11. SUPPLEMENTARY NOTES

Technical Monitor: J. Hollis, Code 930

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified–Unlimited
Subject Category 82
Report is available from the NASA Center for AeroSpace Information, 800 Elkridge Landing Road, Linthicum Heights, MD 21090; (301) 621-0390.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Two papers make up the body of this report. One presents a single-pass adaptive vector quantization algorithm that learns a codebook of variable size and shape entries; the authors present experiments on a set of test images showing that with no training or prior knowledge of the data, for a given fidelity, the compression achieved typically equals or exceeds that of the JPEG standard.

The second paper addresses motion compensation, one of the most effective techniques used in the interframe data compression. A parallel block-matching algorithm for estimating interframe displacement of blocks with minimum error is presented. The algorithm is designed for a simple parellel architecture to process video in real time.

14. SUBJECT TERMS

Data compression

15. NUMBER OF PAGES

16

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited